Slide 1

Administrivia

- Reminder: Homework 6 due Wednesday (but okay to turn in through Friday).

Slide 2

Layers of I/O Software — Recap/Review

- Typically organize I/O-related parts of operating system in terms of layers — more modular.
- Usual scheme involves four layers:
  - User-space software — provide library functions for application programs to use, perform spooling.
  - Device-independent software — manage dedicated devices, do buffering, etc.
  - Device drivers — issue requests to device (or controller), queue requests, etc.
  - Interrupt handlers — process interrupt generated by device (or controller).

  (Figure 5-11 in textbook —- but in my thinking device drivers and interrupt handlers should be side by side.)

Slide 3

I/O Software Layers — Example

- As an example, sketch simplified version of what happens when an application program calls C-library function `read`. (`man 2 read` for its parameters.)

  (Also see Figure 5-17 in textbook.)
- (Want to read all the details? For Linux, source (not current, but representative?) is available in `/users/cs4320/LinuxSource`.)

Slide 4

Sidebar: “Opening” Files

- (This is really kind of part of the discussion of filesystems?)
- You know that in most programming languages you have to “open” a file before working with it. What does that do?
- in UNIX/Linux, ultimately results in making an “open file” system call, which builds a system-specific data structure in the O/S’s memory, adds it to the list of open files for this process, and returns to the program the index into this list (called a “file descriptor”).
- What’s in that data structure? as best I can tell, function pointers for code to perform operations such as read and write. More about these functions soon.

Slide 5

User-Space Software Layer — C-Library `read` function

- Library function called from application program, so executes in “user space”.
- Sets up parameters — buffer, count, “file descriptor” constructed by previous `open` — and issues `read` system call.
- System call generates interrupt (trap), transferring control to system `read` function.
- Eventually, control returns here, after other layers have done their work.
- Returns to caller.

Slide 6

Device-Independent Software Layer — System `read` Function

- Invoked by interrupt handler for system calls, so executes in kernel mode.
- Checks parameters — is the file descriptor okay (not null, open for reading, etc.)? Returns error code if necessary.
- If buffering, checks to see whether request can be obtained from buffer. If so, copies data and returns.
- If no buffering, or not enough data in buffer, calls appropriate device driver to fill buffer (file descriptor indicates which one to call, other parameters such as block number), then copies data and returns.

Slide 7

Device-Driver Layer — Interaction with Controller

- Contains code to be called by device-independent layer and also code to be called by interrupt handler.
- Maintains list of read/write requests for disk (specifying block to read and buffer).
- When called by device-independent layer, either adds request to its queue or issues appropriate commands to controller, then blocks requesting process (application program).

  (This is where things become asynchronous.)
- When called by interrupt handler, transfers data to memory (unless done by DMA), unblocks requesting process, and if other requests are queued up, processes next one.

Slide 8

Interrupt-Handler Layer — Processing of I/O Interrupt

- Gets control when requested disk operation finishes and generates interrupt.
- Gets status and data from disk controller, unblocks waiting user process.

  At this point, “call stack” (for user process) contains C library function, system `read` function, and a device-driver function. We return to the device-driver function and then unwind the stack.

Slide 9

I/O — Device Specifics

- Textbook presents a tour of major classes of devices. For each, it looks first at what the hardware can typically do, and then at what kinds of device-driver functionality we might want to provide.
- Worth reviewing; we will look at a few today. (In reading, okay to skim things not mentioned in lecture.)

Slide 10

Disks — Hardware

- Magnetic disks:
  - Cylinder/head/sector addressing may or may not reflect physical geometry — controller should handle this.
  - Controller may be able to manage multiple disks, perform overlapping seeks.
- RAID (Redundant Array of Inexpensive/Independent Disks):
  - Basic idea is to replace single disk and disk controller with “array” of disks plus RAID controller.
  - Two possible payoffs: Redundancy and performance (parallelism).
  - Six “levels” (configurations) defined. Read all about it in textbook if interested.

Slide 11

Disks — Hardware, Continued

- Solid-state disks/drives — not much in the textbook, but Wikipedia article (usual caveats!) has some details. Executive-level summary:
  - Basic idea is to provide something that to the O/S looks like a traditional disk but without moving parts. Various implementations.
  - Some implementations provide non-volatile storage, but not all do.
  - Lack of moving parts means access times don't include seek time; many implications.
  - Currently faster but more costly and (sometimes?) less reliable. Also some implementations limit number of writes to particular block.

Slide 12

Disk Formatting

- Low-level formatting: Each track filled with sectors (preamble, data, ECC bits).
- Higher-level formatting: Master boot record, partitions (logical disks), partition table. Master boot record points to boot block in some partition. Partition table gives info about partitions (size, location, use).
- Partition formatting: Boot block, blocks for file system.

Slide 13

Disks — Software

- Many devices really pretty much have to be controlled by one process at a time — keyboard, mouse, etc.
- Disks, however, often(?) need to be able to service many processes more or less concurrently.
- So device drivers typically have some way of queueing requests and managing the queue.

Slide 14

Disk Arm Scheduling Algorithms

- A little more about hardware: Time to read a block from disk depends on seek time, rotational delay, and data transfer time. First two usually dominate.
- Typical device driver for disk maintains a queue of pending requests (one per disk, if controller is managing more than one). What order to process them in? several “disk arm scheduling algorithms”:
  - FCFS (first come, first served).
  - SSF (shortest seek first).
  - Elevator.

  How do they compare with regard to ease of implementation, efficiency?

Slide 15

Disk Error Handling

- Almost all disks have sectors with defects. Some controllers can recognize them (repeated failures) and avoid them; if not, O/S (device driver) must do this.
- Other kinds of errors also possible, e.g., failure to correctly position read/write head; also must be handled either by controller (if possible) or O/S.

Slide 16

Clocks — Hardware

- System clock: Can be simple or programmable. Programmable clock can generate either one interrupt after specified interval or periodic interrupts (“clock ticks”).
- Backup clock: Usually battery-powered, used at startup and perhaps periodically thereafter.

Slide 17

Clocks — Software

- Clock(s) can be treated as I/O devices, with device driver(s). Functions to provide:
  - Maintain time of day.
  - Enforce time limits on processes.
  - Provide timer / alarm-clock function.
  - Do accounting, profiling, monitoring, etc.
  - Do anything required by page replacement algorithm (e.g., turn off R bits in page table entries).
- Provide this functionality in code to be called on periodic clock-tick interrupts.

Slide 18

Character-Oriented Terminals — Hardware Overview

- Hardware consists of character-oriented display (fixed number of rows and columns) and keyboard, connected to CPU by serial line.
- Actual hardware no longer common (except possibly in mainframe world), but emulated in software (e.g., UNIX/Linux terminal windows) so old programs still work. (Why does anyone care? those “old programs” include command shells, text editors, etc., which some of us claim are still useful, and likely to be stable.)

Slide 19

Character-Oriented Terminals — Keyboard

- Hardware transmits individual characters one at a time.
- Device driver can pass them on one by one without processing, or can assemble them into lines and allow editing (erase, line kill, suspend, resume, etc.). Typically provide both modes (“raw” and “cooked”).
- Device driver should also provide:
  - Buffering, so users can type ahead.
  - Optional echoing.

Slide 20

Character-Oriented Terminals — Display

- Hardware accepts regular characters to display, plus escape sequences (move cursor, turn on/off reverse video, etc.).

  In the old days, escape sequences for different kinds of terminals were different (yes, really!). A UNIXworld mechanism for coping was to have a `termcap` database that allows calling programs to be less aware of device-specific details.
- Device driver should provide buffering.

Slide 21

Character-Oriented Terminals — Programs

- Many examples of software that uses this kind of device — basically, anything text-based but not line-oriented, for example text editors such as `vim`, `emacs` (in non-graphical mode).
- Libraries for writing such software are system-dependent. One commonly used in UNIXworld is `ncurses`.

Slide 22

GUIs — Hardware Overview

- PC keyboard: Sends very low-level detailed info (keys pressed/released); contrast with keyboard for character-oriented terminal.
- Mouse: Sends (delta-x, delta-y, button status) events.
- Textbook says display can be vector graphics device (rare now, works in terms of lines, points, text) or raster graphics device (works in terms of pixels). Raster graphics device uses graphics adapter, which includes:
  - Video RAM, mapped to part of memory.
  - Video controller that translates contents of video RAM to display. Typically has two modes, text and bitmap.

  High-end controllers (getting more common) may incorporate processor(s) and local memory. (Indeed, they're becoming usable for general-purpose computing — "GPGPU"(!))

GUI Software — Basic Concepts

- “WIMP” — windows, icons, menus, pointing device.
- Can be implemented as integral part of O/S (Windows) or as separate user-space software (UNIX).

Slide 23

GUIs — Keyboard

- Hardware delivers very low-level info (individual key press/release actions).
- Device driver translates these to character codes, typically using configurable keymap.

Slide 24

Slide 25

GUIs — Display (Windows Approach)

- Each window represented by an object, with methods to redraw it.
- Output to display performed by calls to GDI (graphics device interface) — mostly device-independent, vector-graphics oriented.

Slide 26

GUIs — Display (Traditional UNIX Approach)

- X Window System (the pedantic call it that and not “X Windows”) designed to support both local input/output devices and network terminals, based on a client/server model.
- “Clients” here are programs that want to do GUI I/O; “server” is a program that provides GUI services. An “X server” can run on the same system as the clients, a different UNIX system, an “X terminal (where it’s the “O/S”), or under another O/S (“X emulators” for Windows).

  (Figure 5-33 in textbook.)

X Window System, Continued

Slide 27

- Core system is client/server communication protocol (input and display events akin to those in Windows) and windowing system.
- “Window manager” and/or “desktop environment” is separate, as are “widget” libraries.
- Modularity makes for flexibility and portability, at a cost in performance. Some Linux distributions moving toward alternatives (presumably to emphasize performance over flexibility).

Minute Essay

Slide 28

- Next time I’ll say a little more about I/O, mostly how it’s done in Linux versus Windows. Other questions about I/O?